Oskari Holm

Service Integration With External Provider

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

8 May 2018

Metropolia

Author	Oskari Holm
Title	Service Integration With External Provider
Number of Pages	30 pages
Date	21 May 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialization Option	Software Engineering
Instructors	Janne Lumijärvi, Team Leader Ilpo Kuivanen, Senior Lecturer

The goal of this thesis was to create a web application for Dispecto Oy that would integrate an external webservice into a web service they themselves are developing. This application was to also be external and handle all communication between the two services.

The thesis covers the development of that application, including setting up a database for it with SQLite. It explains how the application was created with framework and gives an overview on the basic ideas behind how that framework work. In addition, it also details using Eloquent ORM and its models to work with the database.

As the application created needed to use some resources on Dispecto's service, authorizing it to access them was key part of the development. The thesis looked into how OAuth2 authorization works, paying particular attention to its authorization code flow, as well as mentioning some key security issues to consider when using it.

Finally, the thesis explains how the communication between two services is handled through the application. In addition to a detailed explanation of the processes happening in the application, it also explains how the HTTP requests were created using curl.

The application was successfully finished. However, due to changes in plans it has not actually been deployed yet. A dummy server was created to test it as the external service it was to integrate does not have a test API, and all the tests were successful.

Keywords	OAuth2, Authorization, curl

Tekijä Otsikko Sivumäärä Aika	Oskari Holm Palveluintegraatio ulkoisella ohjelmalla 30 sivua 21.5.2018
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaajat	Janne Lumijärvi, Team Leader Ilpo Kuivanen, Lehtori

Tämän opinnäytetyön tavoitteena oli luoda Dispecto Oy:lle verkko-ohjelma, jonka avulla heidän työn alla olevaan palveluun voitaisiin integroida toinen ulkoinen palvelu. Ohjelman tehtävänä oli huolehtia kaikesta kommunikaatiosta kahden palvelun välillä.

Työ kuvaa ohjelman kehityksen, mukaan lukien SQLite-tietokannan käytön siinä. Se selittää kuinka ohjelma luotiin Slim-sovelluskehyksen avulla ja kuvailee kyseisen sovelluskehyksen pääideat. Lisäksi työ kertoo myös kuinka Eloquent ORM:mää ja sen malleja käytettiin SQLite-tietokannan kanssa.

Koska luotavan ohjelman tarvitsi käyttää joitain resursseja Dispecton palvelusta, oli sen auktorisoiminen käyttämään niitä yksi työn tärkeimmistä asioista. Työ tutki OAuth2-auktorisoinnin toimintaa tässä tarkoituksessa, erityisestin sen authorization code flow:ta, sekä kuvasi joitakin merkittäviä turvallisuusseikkoja jotka pitää ottaa huomioon sitä käytettäessä.

Lopuksi työ myös kuvaa kuinka kommunikointi kahden palvelun välillä ohjelman lävitse toimii. Se siältää tarkan prosessi kuvauksen siitä, mitä ohjelmassa kommunikoinnin aikana tapahtuu sekä lisäksi se kertoo kuinka erilaiset http pyynnöt luotiin curl:ia käyttäen.

Ohjelma saatiin kehitettyä, mutta muuttuneiden suunnitelmien takia sitä ei vielä ole otettu todelliseen käyttöön. Sitä varten luotiin erillinen testaus palvelin, koska integroitavalla palvelulla ei ole omaa testi rajapintaa. Kaikki testit tätä testipalvelinta vasten saatiin suoritettua onnistuneesti.

Avainsanat	OAuth2, auktorisointi, curl

Metropolia

Contents

List of Abbreviations

ORM	Object-relational mapping. The set of rules for mapping objects in a programming language to records in a relational database, and vice versa.
PHP	Hypertext Preprocessor, a server side programming language
DIC	Dependency Injection Container
ACID	Atomic, Consistent, Isolated, Durable
CSRF	Cross Site Request Forgery
TLS	Transpor Layer Security

1 Introduction

Many web services (such as Facebook, Google etc.) allow independent developers to create applications that are able to use the service's resources. It is common to do this by having the user allow the external applications to access certain parts of the service, a process called authorization.

This thesis was done for Dispecto Oy, a professional consulting and software development company, and its aim was to create an external application that would integrate another service to the web service Dispecto and their partner are currently working on (DispectoService). In an older version, the service was integrated directly into DispectoService, but for the current version an external integration was desired. One big reason for supporting the integration of other services via external applications was that other developer could potentially create their own integrations themselves with such applications, freeing Dispecto's developer resources for other tasks. In Dispecto these applications are called providers, and the one being developed for this thesis was to serve as a possible example for other potential developers.

An order management system, the external service, is used to periodically bring certain orders into DispectoService. This will happen via the provider, without any direct communication taking place between DispectoService and the external service. As both services keep track of the orders in real time and are capable of making changes in them, the provider will also cover sending various updates and messages from one service to another. To do all of this, however, the provider needs authorization to access some parts of DispectoService.

This thesis describes the development of the provider, how the communication of the two services works through it and how the provider keeps both services up to date. In addition, it also covers how OAuth2 authorization could be used to authorize these kinds of applications to access DispectoService's resources. This is particularly important in regard to the provider serving as an example for developing other such providers.

2 Project Setup & Application Architecture

Except for a link to authorize it the provider is entirely server side. PHP was chosen as the programming language, not only because it is a server-side language, but also because the older version of the integration was created with it and some parts of it could then either be reused with some modification or simply be used as a base to work from. This part details other components used to set up the provider, such as the framework and database engine used, as well as giving a brief overview of the application.

2.1 Slim Framework

The provider was built using Slim 3. Slim is a fast micro PHP framework that allows its users to quickly write simple yet powerful web applications and APIs. It is fast and powerful, containing very little code, whilst at the same time being far less extensive than many other frameworks such as Symfony or Laravel, making it ideal for this project. Slim documentation describes this simplicity with the following:

> At its core, Slim is a dispatcher that receives an HTTP request, invokes an appropriate callback routine, and returns an HTTP response. That's it. [1.]

2.1.1 Slim Configuration

In Slim the Application (Slim\App class) is the entry point, to the application being created. It accepts just one argument, either a Container instance or simple array for configuring the Application's settings. It registers all the routes and their callbacks and essentially everything works through it.

2.1.2 PSR-7

Slim supports PSR-7 (PHP Standard Recommendation) HTTP interfaces for request and responses. PSR-7 is a set of interfaces defined by PHP Framework Interop Group, used for representing HTTP messages and URIs for HTTP communication. In practice this means that Slim's application routes could return a Guzzle stream instead of a Slim\Http\Response. While Slim offers its own PSR-7 implementations, it is also possible to replace them with any third party PSR-7 implementations. [2.]

2.1.3 Routing

As mentioned earlier all routes in Slim are registered via the Application. Slim does not use automatic URL formulas so it's very flexible as any route patterns can be mapped to any function as the user desires. Routes are most commonly linked to a particular HTTP verb (such as GET), but it would also be possible to link them to more than one verb, though this approach is not used in this provider.

All the route callbacks accept three parameters: Request, Response and Arguments.

The request parameter contains information about the incoming request such as headers, variables it has, and the data it possibly contains. The response parameter is the object to which Slim adds its output and headers, and once complete will be turned into the HTTP response Slim sends out.

The argument parameter is optional and contains named placeholders from the URL. For example, if there was a route with the URL www.someapi/tickets/{id}, the id would be the argument, and you could send a request like www.someapi/tickets/42, easily getting the desired ticket's id of 42 from the argument.

2.1.4 Dependency Injection Containers

Slim uses a dependency container to prepare, manage, and inject application dependencies. [3.] Dependency injection is a design pattern where the dependencies of one object are provided to it by another object. So instead of creating references to the other objects and services it works with by itself, they will be given to them at creation time by some external entity. The idea is to configure the DIC so that it will be able to load the dependencies the application needs at the time it needs them. Once a dependency is created, the container will store it, allowing it to supply it again if needed later (instead of creating a new instance of the service).

The container is created by having the Application call a getContainer() function. Once it is created, services can then be added to it by defining a function that creates an instance of the service as shown in listing 1 below. On the first time the application tries to access

the dependency this will be called, and the service will be created. On the following time the same instance that was created on the first call will be returned instead.

```
$container['myService'] = function ($container) {
$myService = new MyService();
return $myService;
};
```

Listing 1. Adding a service to the DIC

2.1.5 Slim Error Handler

Slim comes equipped with a very simple default error handler. It simply sets the status code of the response to 500 and content type to text/html and also appends a generic error message to it. While this is very basic, and for example wasn't enough for the provider, it is possible to define custom error handlers.

Defining the error handler is similar to adding a service to the DIC, but instead of creating and returning the service the error handler's function must instead return an instance of Psr7's response interface.

While the provider required a more complex implementation of the error handler than the default one offered by Slim, it still remained rather simple as seen in Listing 2. The main problem with the default handler was that the status code always remains as 500 regardless of what Exception was being thrown, and the error message is generally useless. Instead, the message is replaced with the actual error message and a trace of the Exception is added to the response as well. In addition, the actual error code of the exception is set as the response's status code, unless it was below 100 or above 599 in which case the 500 is still used.

```
$container['errorHandler'] = function ($container) {
return function ($request, $response, Exception $exception) use ($container) {
$data = [
'code' => $exception->getCode(),
'message' => $exception->getMessage(),
'trace' => $exception->getTrace()


];
$code = $exception->getCode();
if ($code < 100 || $code > 599) {
$code = 500;
}
return $container->get('response')->withStatus($code)
->withHeader('Content-Type', 'application/json')
->write(json_encode($data));
};
};
```

Listing 2. Defining a custom error handler

2.2 SQLite

The provider features a database for storing the requests coming from both External-Service and DispectoService as well as storing its own internal state. SQLite is the most widely deployed database in the world that features in, for instance, every Android, iPh-one, and iOS device. It is also, as the name suggests, a SQL database engine and it was used with the provider for the following reason:

> SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. [4. Quote]

Where most database engines use a separate server process to operate on and also send and receive request to the database via TCP/IP protocol (client/server architecture), SQLite uses a serverless architecture instead. It does not require and instead writes and reads directly to and from ordinary disk files. The key difference, as shown in Figure 1 is that the application communicates directly with the application instead of a process that is on the same server as the database. A single file contains a complete database with its tables, views, indices etc. The main advantage of this is that there is no separate server process to install, configure and maintain. The disadvantage is that it is more prone to bugs in the application that uses it. Where a wrong pointer would not cause any harm on a server, it could corrupt data on a disk file.

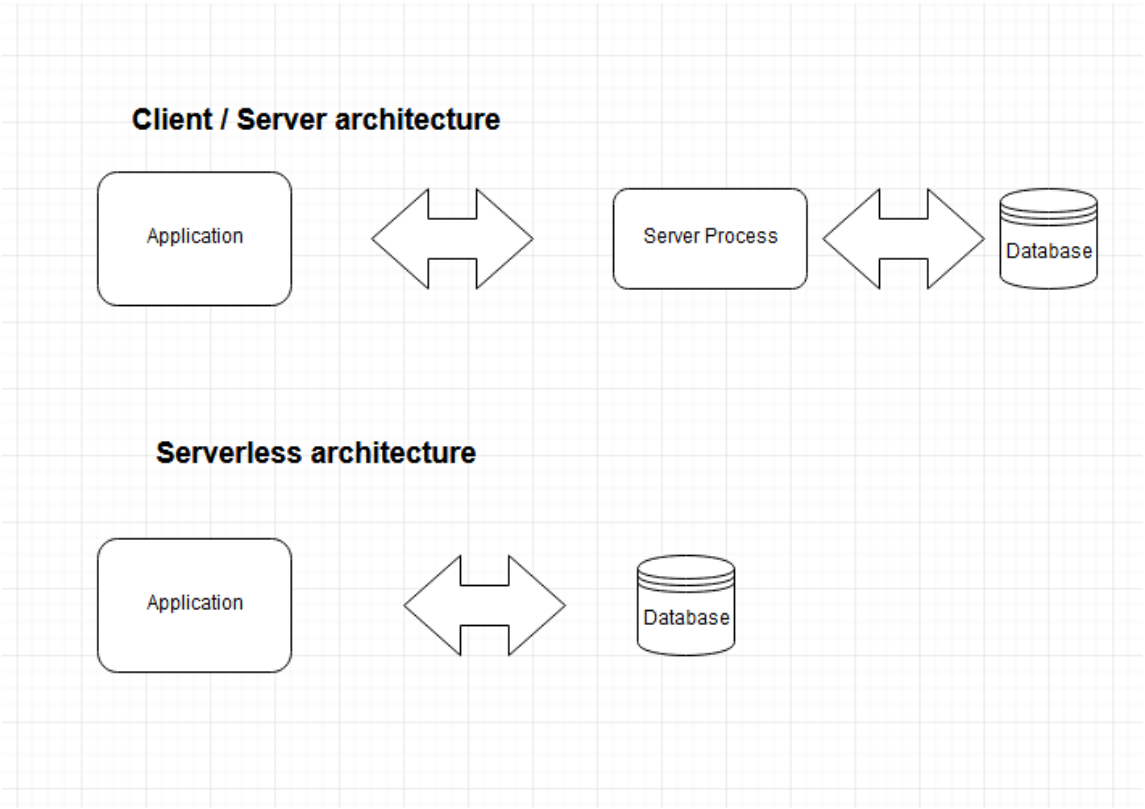

Figure 1. Client / Server vs serverless architecture

Being self-contained means that SQLite is a compact library that requires only limited support from operating systems and has very few dependencies. In fact, it runs on all operating systems, even on the most limited embedded operating systems, and only uses a few standard C-library calls. Even with all features its library size can be below 500KiB, though this depends on the target platform and optimization [4.].

SQLite implements serializable transactions that follow the ACID principles, even if the transaction is interrupted by things like power failures or program crashes, making it a transactional database. Either all the changes in a single transaction happen or none at all and SQLite has an extensive test suite for checking this.

For the provider this makes SQLite ideal as it does not require setting up another server process for the integration's database, and instead allows for the entire database to simply be shipped with the service. The zero-configuration means that it requires no installation or configuration of any kind and could immediately be taken into use without any time being lost on setting it up.

2.3 Eloquent ORM

Eloquent is an object relational mapping tool (ORM) that is part of the Laravel PHP framework, but it can be installed and used without it which is the approach used in this project.

2.3.1 Eloquent Models

In Eloquent every database table has Model representing it which is used to interact with the corresponding table. An Eloquent Model is defined by creating a class that extends Eloquent's Illuminate\Database\Eloquent\Model class.

By default Eloquent uses a snake case plural of the class name as the table name (so for example, for a class named ”User” the table name would be ”user”), but this can be overwritten by explicitly specifying $table variable.

Eloquent also automatically assumes that the table has updated_at and created_at columns by default. However, using these is not mandatory and as neither of these were used in the provider's database tables, they were simply disabled by setting $timestamps variable of the various models as false.

```
$class Update extends Model
{
protected $order;
protected $table = 'ext_updates';
public $timestamps = false;
```

Listing 3. Defining an Eloquent Model

2.3.2 Using Eloquent

Once the Eloquent Models are defined, the database can then be queried by using the many methods Eloquent provides instead of writing direct database queries. For example, to get all the records in the ext_updates table, the all() method would be called.

```
$allUpdates = Update::all();
```

Listing 4. Retrieving all records from ext_updates

This method (and all other Eloquent methods) look similar to static methods. However, they are not and what is actually happening is that Eloquent offers a façade, allowing for methods on an object to be called as if they were declared statically. The actual (non-static) method is found in the Eloquent Model class that the model extends.

The different methods can also be combined and chained to create more complex queries. For example, in listing 5 an Update table is queried for entities whose message_id matches the given $id and where their comm_id is in the $comms array and their halted value is 0. Also, only columns id (with alias num), comm_id, and message_id are selected, and finally the found records are grouped by comm_id and message_id.

```
$result = Update::where('message_id', $id)
->whereIn("comm_id", $comms)
->where('halted', 0)
->select("id as num", "comm_id", "message_id")
->groupBy('comm_id', 'message_id')
->get();
```

Listing 5. Chaining Eloquent methods

While Eloquent offers a wealth of methods to use in building the queries, it is of course possible that some queries prove too complex to replicate. For that reason, Eloquent offers a selection of Raw Methods (such as whereRaw, selectRaw, etc.). The whereRaw method for example takes a raw where clause as its first argument. The raw statements will be injected as query strings, so care should be taken not to create SQL injection vulnerabilities when creating them. [5.]

2.4 Phpdotenv

Application configurations usually vary between different environments or deploys (such as staging, production, developer environments, etc.). These often include things like database user names and keys, credentials for other services. As such storing them as constants directly in the code would be a major security flaw.

One common approach is to store them in files that are not included in version control, and while it is vastly better than including them directly in the code it still has several issues ranging from simply accidentally checking the config files into service control to format issues.

Another way is to store the configurations in environment variables instead, which is also one of the tenets of a twelve factor application. [6.] The key advantage to them is that they are easy to change between different environments and stand little chance of being accidentally added to version control.

Phpdotenv is a PHP version of the original Ruby dotenv by vlucas and it loads variables from .env to PHP's getenv(), $_ENV and $_SERVER automatically. The environment variables are stored in the .env file as strings:

```
S3_BUCKET="dotenv"
SECRET_KEY="super_secret_key"
```

Listing 6. .env variable examples

The .env file should never be put into version control and should instead be added to the projects .gitignore so that it never ends up as being accidentally committed. Instead an .env.example file with all the required variables defined in it, expect for sensitive values such as database keys etc. This file is added to version control, and is then copied for to a local .env file for each environment and filled out.

The .env file is placed in the project root and is then simply loaded in the project with:

```
$dotenv = new Dotenv\Dotenv(__DIR__);
$dotenv->load();
```

Listing 7. loading the .env file

The variable can then be accessed with PHP's own getenv(), for example.

Using the .env files in this way not only ensures that no passwords or other sensitive values will end up in the version control history, but also allows for easily changing the values between different environment, and prevents sharing production values between all collaborators in the project.

3 Application

The provider is essentially composed of three main parts, each with its own namespace. At the top there is the main Slim Application that handles the routing and dependencies,

and two manager classes to help it in this. Then there are the ExternalService and DispectoService Integrations and their associated classes to handle the request and responses coming from and sent to the two services. Both are added to the application using the DIC, and then called as needed.

Finally, in addition to these the provider uses several external libraries, including the already introduced PHPDotenv, Eloquent and Slim libraries.

3.1 Main Application

While the Slim Application works as the main entry point to the provider, loading all the required dependencies and routes and their callbacks it does not actually perform any of the integrating itself. Instead it simply gets the needed Integrations from the dependency container and either forwards the incoming requests to one of two manager classes: Events- and ActionsManager or in the case of authorization simply forwards it directly to the DispectoService Integration.

The two manager classes are similar to each other, and their role is to actually look at the data the requests contained and select the appropriate action based on it as well as ensuring that they are responded to correctly (i.e. in a way that DispectoService understands).

3.2 DispectoService Integration

The DispectoService Integration is a single class that contains functions for creating and sending all the various messages sent from the provider to DispectoService. In addition, it also features methods for authorizing the provider on DispectoService's own API, and checking that incoming requests from Siriwise have the correct authentication details. The Slim Application calls it directly when authorizing the provider on DispectoService, and passes it along via the Event- and ActionManager when making requests to or from ExternalService.

3.3 ExternalService Integration

More complex than the DispectoService part, this namespace contains the classes to handle integrating the ExternalService as well as the database model classes. The most important class here is the ServiceInfoIntegration whose method's the Events and ActionManager classes call. It then processes the corresponding requests, with the help of other classes in the namespace. Many of these are abstract classes or interfaces that define the methods the actual classes must have, or simple helper classes to, for example, parse an xml the ExternalService responded with.

The ServiceInfoIntegration is essentially the middleman between ExternalService and DispectoService, calling on DispectoService to retrieve the resources required for the current ExternalService updates, and then passing the data required on to ExternalService.

Another important class in this namespace is called Service, and it is the class ServiceInfoIntegration calls on when it has retrieved all the data it needs and is actually creating and sending the requests to ExternalService.

3.4 Database

The SQLite database used for the is mainly used for logging data about the requests and messages to and from the provider and contains several tables for this. In addition to simply keeping a log about all the incoming and outgoing requests, this means that all updates do not necessarily have to happen immediately. Instead they can be stored in the database and run later, which is useful both in the case that errors occur during the updates, and in reducing the load when a request contained several expensive update operations.

Some data also have different values in DispectoService and ExternalService whilst still holding the same meaning. For example, an order's status has a slightly different description in DispectoService than it does in ExternalService. The database has several tables for mapping the values correctly to both services.

4 Authorization with OAuth2

In order for the provider to be able to create and update orders in DispectoService it needs access to DispectoService's API. Of course, it should not get full access as third party application and should only be able to access the features it needs to. OAuth2 (Open Authorization) is an authorization framework for allowing such limited access to a web service.

> OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices. [7. Quote].

Though often mistaken as the same, authorization is not the same as authentication. Authorization essentially means the permission that users have, for example allowing some users of an application to view and edit data on the application, while others are only authorized for a more limited access that only allows viewing. Authentication on the other hand is the verification that someone is who they claim to be, i.e. login and password. While there are many implementations of OAuth2 that use it for both it is not intended for authentication, only for authorization.

4.1 Application Roles

In OAuth2 there are four defined Roles:

- **Resource Owner** is the user who authorizes an application to use their account. In the provider's case this would be the DispectoService user that authorizes the provider to use DispectoService.
- **Client** is the application that wants the access, and which must first be authorized to do so. i.e the provider that was created for this thesis.
- **Resource server** is where the user accounts are hosted, and **Authorization server** is the server that verifies the user and provides the access tokens for the application. Essentially, they are the service's API.

4.2 Authorization flow

In OAuth2 the processes for authorization are called as flows. The general flow of OAuth2 authorization goes as follows:

- 1. The client requests access to resources from the resource owner.
- 2. If the resource owner authorizes the request, it then receives an authorization grant.
- 3. With the grant the client requests an access token from the authorization server. It identifies itself on the server with the client credentials and the grant it just received.
- 4. The server authenticates the client with the credentials presented, and then issues an access token to it.
- 5. With the access token, the client then requests for resources on the resource server.
- 6. Provided the access token is valid, the server responds with the requested resources.

However, while the above describes how the OAuth roles interact with each other in general, the actual flows vary depending on the grant type. Oauth2 has four different grant types, each used for a different purpose.

4.3 Application Registration

When using OAuth the service that the application uses will need to know the application, i.e it needs to registered with the service. The minimum information the service needs are the application name, its website, and its redirect URI or callback URL (though naturally the developers of the application will likely need to describe other details about their applications as well).

Once the application is registered on the service, the service will issue client credentials, formed of two parts: client identifier and client secret. The client identifier is simply a string that is used to identify the application on the service. It is public and is used in the authorization URLs. The client secret however, is used to authenticate the application on the server (essentially a password) and is therefore private between the two.

4.4 Grant Types

Authorization code grant type is used with server-side applications and is the most commonly used grant type. As a redirection based flow this requires the application to be

able to act with the user agent (web browser). It is also the grant type the provider uses and works as follows:

First a user visits the website where they want to log in, that in the provider's case is simply a blank web- page with a link to authorize. However, the user's identity is third party and actually stored somewhere else such as Google, Facebook, or in our case, DispectoService. As the user clicks the log in link they are then redirected to the site where they are asked to accept some permissions such as the prompt in figure 2.

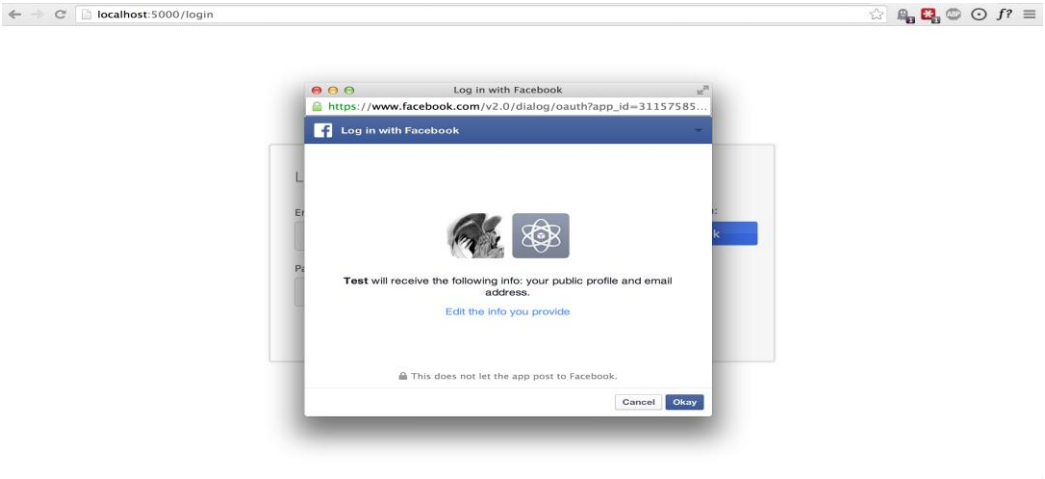

Figure 2. Example of permission prompt on facebook

In case they accept the permissions and log in with their third party credentials, they will then be redirected back to the application (the provider) with an authorization code. The application will then make a request to the third party with the authorization code provided, receiving an access token in return. This access token is what is then used to actually retrieve the user's information, and in the provider's case to also authorize the other requests made to DispectoService. This entire flow is visualized in figure 3 below.

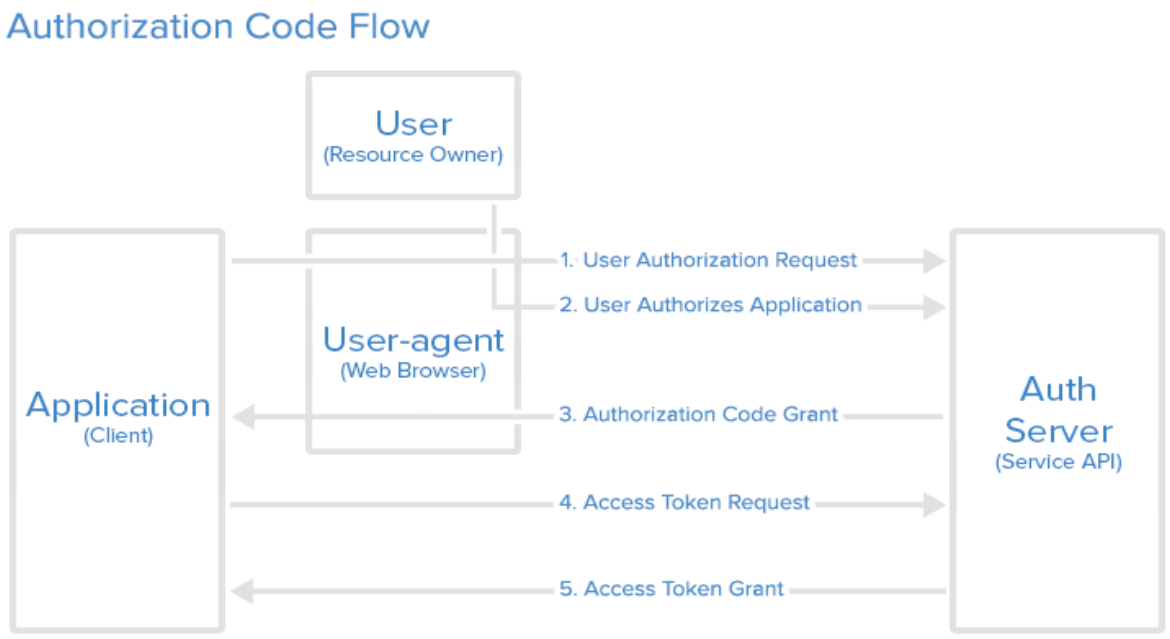

Figure 3. Authorization Code flow [9.]

When the initial access request is made it is also possible to specify the level of access that is desired which is done by specifying a scope parameter in the request URL. For example, as the provider will have heavy interaction with DispectoService's orders, it will have orders as its authorization request's scope. This will allow it to access the orders in DispectoService, but not anything else.

```
$URL = $authUrl . "?client_id=$clientId" . "&re-
sponse_type=code&state=$state&scope=orders";
```

Listing 8. Specifying orders as the desired level of access

Implicit grant type is used with mobile applications or client-side web applications (i.e. applications that don't have a server side component). This is because when using it there is no need to store any secret key information at all. With this grant type you simply log in with a third party identity, and receive the access token with that log in.

Password Credentials is used with trusted applications, such as applications that the service owns itself. It should only be enabled if the other flows are not viable [8.]

The key thing about password credentials grant type is that when using it the user provides their credentials directly to the application, which then uses them to obtain the

access tokens. This means it should never be used with third parties, as they could easily collect the credentials and store them.

Client Credentials is used with application API access when requesting access to protected resources under its own control (i.e there is no third party).

4.5 Refresh Tokens

An access token usually expires after some time, resulting in “Invalid Token Error” errors when requests are made with it. However, it is possible for the API to provide a refresh token when the original request was first responded to. The refresh token can then be used to request a new access token, preventing the need to constantly log in again.

4.6 OAuth2 security

On their own page Oauth2 highlights three known security issues: Phishing attacks, Clickjacking and Redirect URL manipulation [9.]

4.6.1 Phishing

In regard to Oauth2 phishing attack means a case where the attacker creates a web page that looks identical to the actual service’s authorization page, and then tricks the user into visiting the page. To help prevent attacks of this kind the authorization server should use Transport layer security protocol (TLS), i.e HTTPS in practice, and consider using whitelisting.

Phishing represent a very real security issue. In 2017 over a million Google accounts were successfully targeted by an OAuth based phishing attack. The targeted users received a phishing email into their Gmail account. The email claimed that one of the user's contacts had shared a document with them and included a button that genuinely took them to a Google page. When they logged in, a service called Google Apps prompted them to give access to the user's email account, contacts and documents. If the

permission was given, the service used it to send further copies of the same email to everyone in the user's contacts. [10.]

4.6.2 Clickjacking

Clicjacking is an attack where the attacker creates a website that loads the authorization server's URL in a transparent iframe. This transparent iframe is then placed carefully below the actual page so that when the user clicks on the visible authorization button the invisible button created by the attacker is clicked instead, granting access to the attacker's application instead of the one users think they are authorizing. To prevent this the authorization URL should only be loaded directly in the browser and not in an embedded iframe. The authorization server should always return an X-Frame-Options header with DENY or SAMEORIGIN value, either preventing the page from being displayed in an iframe or only allowing the page to be displayed in a frame on the same origin as the page itself respectively. [11.]

4.6.3 Redirect URL manipulation

In redirect URL manipulation the attacker creates an authorization URL from client id of a known application whilst setting the redirect URL to something under their control. If the redirect URLs are not validated on the authorization server, the attacker can request a token response type and receive an access token when they return to the attacker's application. This can be prevented by having the authorization server validate the redirect URL. The application should register one or more redirect URLs with the server and the server should then only redirect if the redirect URL is an exact match of one of the registered URLs.

4.6.4 Cross site request forgery

In addition to these CSRF attacks can also be a severe issue with OAuth2. The Internet Engineering Task Force (IETF), an organization that is responsible for promoting and developing Internet standards, has created an extensive list of various security issues in their OAuth 2.0 Threat Model and Security Considerations draft, and it has the following to say about CSRF:

> An attacker could authorize an authorization "code" to their own protected resources on an authorization server. He then aborts the redirect flow back to the client on his device and tricks the victim into executing the redirect back to the client. The client receives the redirect, fetches the token(s) from the authorization server, and associates the victim's client session with the resources accessible using the token. [12.]

To protect against CSRF the client and the authorization should both implement countermeasures against it. This is done by including a state parameter in the redirect URL the client sends to the authorization server. The parameter should be unguessable and stored securely, for example a hash of the session cookie. This state will then be sent back to the client with the authorization code, untouched, and checked that it matches.

4.6.5 Other issues

These are not the only security considerations, and simply using OAuth2 by itself will not lead to a secure authorization process. The implementation of the OAuth2 has to take security issues into account. In addition, OAuth2 was designed to provide authorization without providing authentication, and misusing it for authentication as well as authorization would be a major security flaw.

Finally, the two different versions of OAuth, 1.0 and 2.0 should not be confused with each other as they are not compatible. Oauth 1.0 is actually more secure due to the later version's focus on developer simplicity and is a protocol instead of framework. This focus means that the OAuth 2.0 does not directly support encryption, signature or channel binding. While this is intentional, and implementations of the framework are expected to use other outside protection for this, such as TLS, it is also one of the major criticisms of OAuth 2. [13.]

5 Communication

This section will deal with the communication between the provider and the two other services. While communication over the web is a very large subject, this part simply describes the use of CURL to send requests, and detail show the provider communicates with DispectoService with it.

5.1 Php libcURL & CURL Class

CURL stand for Client URL and is a command line tool for transferring data with URLs. LibcURL is a multi-protocol file transfer library created by Daniel Stenberg, that PHP supports. It enables connecting and communicating to different kind of servers, with different types of protocols (such as HTTP, HTTPS, FTP, etc.). In essence it is a way to use URLs directly from code, using the cURL functions.

Using the cURL functions can be slightly complicated, however. PHP CURL Class is a set of PHP classes by Zach Borboa which provide their own functions to make HTTP requests easier. These functions themselves still use the standard, cURL functions and are just a tool to make developing easier. [14.]

5.2 Sending requests to DispectoService

Whenever the provider is making a request to DispectoService, whether to get some resource from it or to create a new one, it will first call the DispectoService/API's createRequest method. This method creates an instance of Request class and sets its bearerToken from the OAuth 2 accesToken that is first retrieved from the database and decrypted. The method will then return the created Request instance.

The Request instance is then used to call DispectoService/API's dispatch method, together with the type of the function and the DispectoService URL that the request will be sent to. The function type is case one of the request types (GET, PUT, POST) as a string, and is actually used with the Request in the dispatch method. The method calls PHP's call_user_func_array to call one of the Request class function which in this case is either get(), put(), or post(). These will then call the Request own dispatch, with the corresponding request type, and the URL that was given, and in post's case also set the post request's mimetype as well.

The dispatch method of the Request class uses the PHP CURL Class for actually sending the request to DispectoService. It first creates an instance of the CURL Class and then set its timeout value. Then it creates a headers array from the parameters it received. The Content-Type header is set to the mimetype that the calling function passed

as a parameter, defaulting to application-json if no mimetype was given. Then it uses the already set bearerToken as the Authorization: Bearer header, provided that the bearerToken was set (it is not set when actually fetching the accessToken from DispectoService). The headers array is then merged with the default headers of the Request class and then passed to the CURL Class with the setHeaders method. Finally the request is sent with the CURL Class get(), put(), or post() methods.

```
$cURL = new CURL();
$cURL->setTimeout($this->getTimeout());
$headers = [
'Content-Type' => $mimeType
];
if ($this->bearerToken) {
$headers["Authorization: Bearer"] = $this->bearerToken;
} else if ($this->username && $this->password) {
$cURL->setOpt(CURLOPT_USERPWD, "{$this->username}:{$this->password}");
}
if ($this->headers) {
$headers = array_merge($headers, $this->headers);
}
$cURL->setHeaders($headers);
if ($requestType === 'POST') {
$cURL->post($URL, $data);
} else if ($requestType === 'PUT') {
$cURL->put($URL,$data);
} else {
$cURL->get($URL, $data);
}
$result = $cURL->rawResponse;
$responseCode = (int)$cURL->httpStatusCode;
```

Listing 9. Sending a CURL Class request

The httpStatusCode is then taken from the CURL Class' rawResponse and php object created from it along with the jsonDecoded rawResponse and the headers used for the request. This object is then returned to the DispectoService/API's dispatch. If the response had no error code for its status, the results are then passed back to whatever method called the createRequest() in the first place.

If there was an error code, however, the error is either thrown with that code and the error message that came along with it, or in case it was a 401 Unauthorized error and the result's code also matched with DispectoService's Token Expired code, the refreshToken() method is instead called to get a new accessToken. If an accesToken is successfully retrieved then it is set as the Request's bearerToken, before sending the request again.

5.3 Receiving requests from DispectoService

When the provider receives request from DispectoService it performs a simple authentication by verifying that the user's webhook secret and tenant id are correct. It first takes the values from the request's data and retrieves the SirwiseAPI. The values are then passed to the SirwiseAPI's authenticate method where they are checked against matching environmental variables (defined in .env and retrieved with getenv() method) and if everything checks out, the tenant property of the SirwiseAPI is set from the received tenant value.

6 Fetching updates from ExternalService

This section will cover the way the provider will actually handle importing the order from ExternalService to DispectoService, updating already existing ExternalService orders in DispectoService, as well as sending updates from DispectoService to ExternalService, both automatic ones and those created manually by DispectoService users.

6.1 Retrieving updates

To use ExternalService orders need to be imported from it to DispectoService. This is done on a regular basis by calling a GET endpoint on the provider, called 'fetch'. The endpoint is used not only to import orders, but to retrieve all other new messages related to orders that have already been imported. The endpoint callback will then get the ServiceInfoIntegration from the dependency container, and call a fetchAction method on it, with the parameters passed along with the request, such as password, account id, and whether to import orders or not (importing orders is an expensive operation, and it may not be done on every call).

After validating the password, the provider will then query the database with the account id provided and check that the account's enabled status has been set to true. In case no account is specified, it will instead get all accounts that have been enabled from the database.

Then an instance of the helper class Service is created for each account and new messages fetched with its fetchNewMessages method. This first creates an instance of FetchMessagesRequest class and sets the request system_id and credentials (login and password). As ExternalService both returns its responses and expects requests to it to be in XML, this class creates one with PHP's inbuilt DOMNode class and sets the element values.

```
<messages_request system_id="c141a354-13b7-4745-8702-4322b6d00388">
<login>dummy</login>
<psw>test123</psw>
<created_date>2017-07-18T16:10:32</created_date>
</messages_request>
```

Listing 10. Example of a fetch Messages request ExternalService expects

Next the request is actually sent to ExternalService with the sendRequest method of the Service class. This method is used for all requests sent to ExternalService, whether they are requests for data from ExternalService or posting updates. To do this it takes an instance of AbstractRequest class as its first parameter, the type of the class it should respond with as second, and finally a boolean to check whether the request is a resent one or not. The idea is that the different types of requests all extend the AbstractRequest that is an abstract class and, for example, defines that all classes extending it must have a getMethod method that returns the type of the method the request is sent with (GET, POST). This way a single method can easily be used for sending the various request.

The actual sending is done with the PHP CURL Class but before it an instance of RequestLog class is created to store the request in the database. Its fields are filled from the passed AbstracRequest's data (for example, using the getMethod mentioned before). This same data is then used to set the required values of the CURL class, such as the request URL and method, it's body and also how long to wait until a timeout error is thrown. The RequestLog's requested_at field by calling the CURL Class' beforeSend method. This method takes a function as its parameter, in this case a simple function that set the RequestLog's requested_at field by calling PHP's time() function. The function will then be called right before the request it sent.

```
$URL = $request->getURL();
$body = $request->getBody();
$method = $request->getMethod();
$log = new RequestLog();
$log->account_id = $this->account->id;
$log->type = $request->getType();
```

```
$log->URL = $URL;
$log->method = $method;
$log->request_body = $body;
$errors = [];
$cURL = new CURL();
$cURL->setTimeout(120); // 2 minutes
$cURL->beforeSend(function () use ($log) {
$log->requested_at = time();
});
//Sending request here
if ($method === 'GET') {
$cURL->get($URL, $body);
} else {
$cURL->post($URL, $body);
}
```

Listing 11. Creating the RequestLog and CURL Class request

Once sent the same PHP CURL class is also used to check the response External-Service sends back. It checks the httpStatusCode and whether there were HTTP errors. In case there were errors, the RequestLog's code field is set to match the error code, and the info field is filled with the error messages. Otherwise the code is set from the response's httpStatusCode and the info field is left empty. In either case the RequestLog's responded_at field is set from the current timestamp and the response_body from the rawResponse of the response, and the log is then saved into the database.

Provided that saving the RequestLog into the database is successful, storeData method is called. Like the sendRequest method it takes an instance of an AbstractRequest and the type of Response as its parameter, as well as the RequestLog that was created. The method gets the actual packages and messages from the response ExternalService sent and in this case creates the corresponding Models and stores them in the database. Finally it calls onMessageReceived method of the UpdateManager class to check if the messages received were duplicates of already processed updates, or in the case they are new to store a new Update entry in the database.

6.2 Processing Received Updates

Once the message is received and the necessary database updates have been done, it's time to actually process the received updates. This is done via the UpdateManager class which first gets all Updates belonging to the account currently being used, that haven't been halted or processed yet. The reason that the response from

ExternalService is not used directly is that there may be older Updates in the database that haven't yet been processed, and those need to be handled too.

```
$result = Update::
join('ext_requests', 'ext_updates.request_id', '=', 'ext_requests.id')
->whereIn('ext_updates.type', $types)
->where('ext_requests.account_id', $account->id)
->whereNull('processed_at')
->where(function ($query) {
$query->where('ext_updates.halted', 0)
->orWhere('ext_updates.force', 1);
})
->orderBy('ext_updates.type', 'asc')
->get(['ext_updates.*', 'ext_requests.type as request_type']);
```

Listing 12. Getting the Updates from the database

The updates will then be processed depending on their type, of which there are five. Though there are differences between all of them, the largest is between an order type and the other types.

6.2.1 Orders

If the Update's type was order, it means it is an order that is to be imported into DispectoService. After retrieving the corresponding order message, the order's id and reference are taken in order to check if the order already exists. A request is sent to DispectoService to query its own database for orders with matching identifier or reference. If any results are received the Update model is then saved with its halted field set to 1 and the info field set with a corresponding message that tell why the update was halted. The same are done if DispectoService instead responds with an error, with the message set into the info of course reporting that an error occurred instead. In both cases the processing for that update is stopped, and the update processing is then continued with the next Update if any are still remaining.

If the order did not exist in DispectoService yet, a new order is then created from the data ExternalService sent. However, though the XML message contains all of the order's data, it cannot be simply directly sent into DispectoService in a single request, as some of the data are actually their own entities in DispectoService. For example, practically every order has a device that is its own class, with its own database documents, and to

which the Order in DispectoService only has a reference to. This means that either a new entry for device either has to be created in DispectoService as well or, in case the same device already exists in DispectoService, fetched from it.

To do this, a DispectoService order is first generated from the XML, using only the data for the fields it has directly. Before a request to create the order is sent, a DispectoService device is also generated from the XML. This is then used to send a POST request to DispectoService to either look for the device or create one in case such a device is not found. The device has either an identification number that is used as a query parameter for the request, with the rest of the device's data in the POST method's request body.

If for some reason an error is encountered while looking for or creating the device, this is again logged in the Update's info field. However, regardless of whether looking for or creating the device was successful or not, another request is sent to DispectoService, this time to create the order. If the device creation failed, a note about the failure will be added to the order and a later update will try to create the device again. This request is sent with the POST method, this time with the order's data in the request body.

If creating the order fails, processing the update is immediately halted and an error message saved into the Update's info field. Otherwise, processing the order continues. First, if the device was either successfully created or found, a PUT request is sent into DispectoService, to update the newly created order's reference to the device to match the correct one. Next, depending on the data received, other documents such as files, or notes are generated from the XML. These are then created in DispectoService with similar POST requests and added to the order with PUT requests. If any errors are encountered with these requests they are reported by saving a message into the Update's info field but processing the update itself is not halted.

Finally, once everything has been generated and added to the order, the Update is saved and marked as processed with a timestamp from the current time, and a message added to its info field noting the created order's id in DispectoService. Then the next update is taken to be processed. Figure 4 depicts an example the above process for importin orders where the device needs to be created.

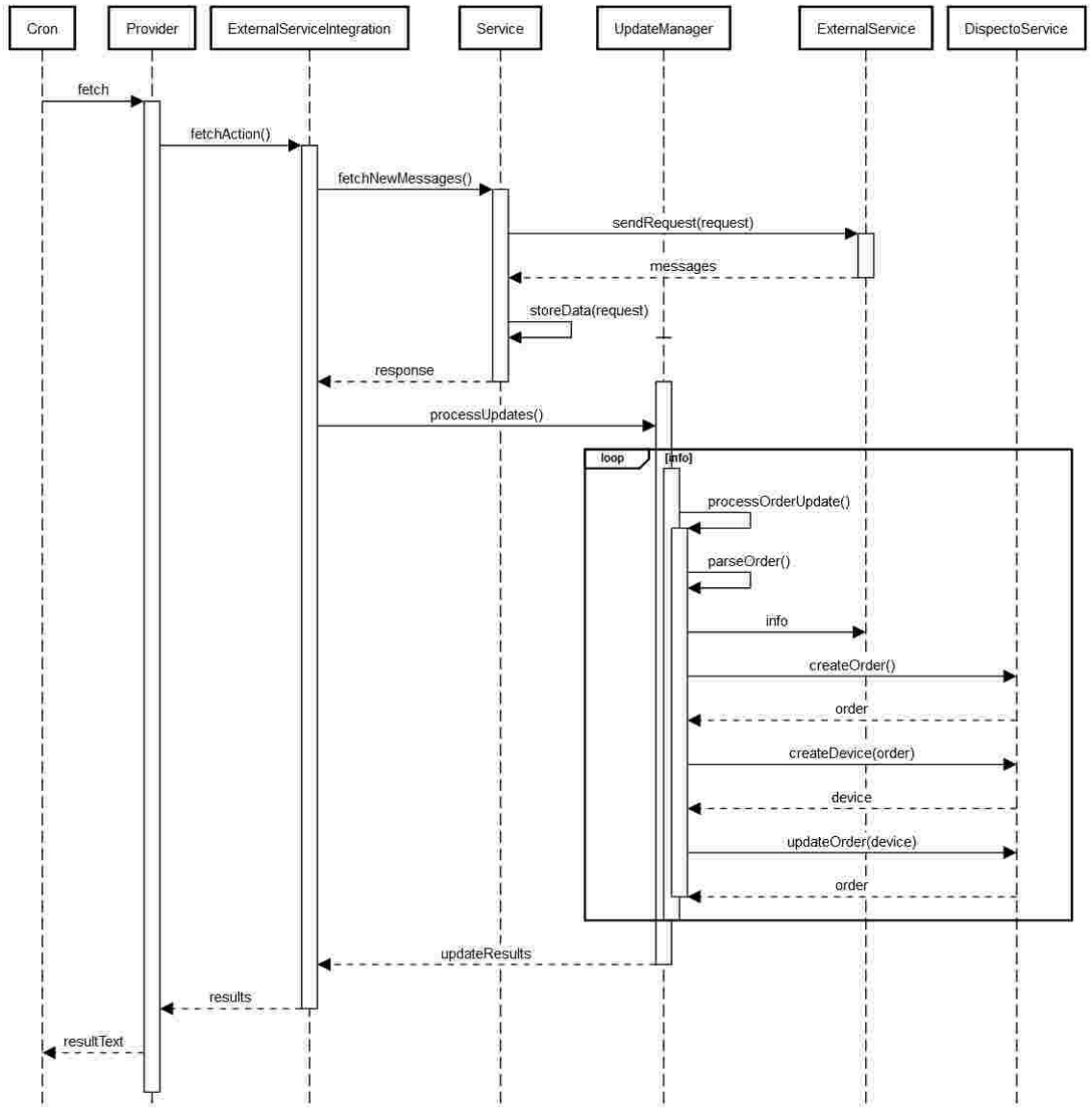

Figure 4. Sequence diagram for importing orders

6.2.2 Other updates

As all the other updates are related to a ExternalService order that has already been imported into DispectoService, the first thing to do when processing them is naturally to get the relevant order from DispectoService. This is done by sending a GET request to DispectoService, with the relevant order's id. If the requests respond with an order the update is then halted and an error message logged like with processing order messages.

If no order is found the update in question is stopped. However, it is not set as halted. The order might simply not have been created yet, and another message to import it could be found even in the same array of updates. Instead a “Waiting for the order to be created” message will simply be saved into the info field of the update before skipping into the next update.

7 Sending updates to ExternalService

In order for the orders to stay up to date in both services, whenever an action is taken in DispectoService that changes something that both services keep track of an update needs to be sent back towards ExternalService. This happens in one of two ways. Either an event automatically triggers an update to ExternalService when certain actions are taken, or alternatively the DispectoService user can manually create certain types of updates.

7.1 DispectoService Event Updates

Some actions in DispectoService, such as changing an order's status, trigger automatic events. In the case of ExternalService orders these events send an update to External-Service. These event updates are designed not only to send data to keep the orders up to date in ExternalService, but also to prevent the actions fully completing in Dispec-toService in case the update to ExternalService was unsuccessful.

When DispectoService sends these updates, it sends them to the '/events' endpoint on the provider via a POST request. The endpoint's callback function first parses the request data with Slims getParsedBody() method. It then retrieves SirwiseAPI from the depend-ency container and uses the data to check the credentials of the request. If the creden-tials checkout it then also gets the ServiceInfoIntegration and the EventManager from the DIC. The request data, its query parameters, and both SirwiseAPI and the ServiceIn-foIntegration are the passed as a parameter to the EventManager's handleEvent method.

The EventManager takes the event's name from the query parameters and selects the correct action based on it. If the event name does not match with any of the events, it

simply returns a string:” Unknown event”. Otherwise it calls the corresponding method from ServiceInfoIntegration and passes the request data as well as the instance of the SirwiseAPI to it as the method's parameters.

The actual implementation of these methods varies depending on the event in question, but two things are common for them all. First, they retrieve the order in question from DispectoService by calling the SirwiseAPI's getOrder() method with the order's id that came with the request's data. Once it is retrieved they validate that the order is in a state where updates regarding it are allowed. If the validation fails or for some reason no order was retrieved, the method returns null to EventManager that then throws an Exception with a message that the order wasn't valid or found.

If the validation succeeds, the way the method proceeds vary, but regardless of which event is in question, each creates an instance of the Service class. They then use this class to create the update to ExternalService that the event requires and send it with the class' sendMessages() method. This method takes the created message as its parameter and creates logs the messages and the request it is about to send to ExternalService to the database, before calling the sendRequest method to actually send them.

Some of the event methods also specify a createLogNotes parameter for the sendMessages() method, in which case it attempts to create logs in DispectoService as well, and in case it fails logs the failure into the database entry of the request.

The response object of the sendRequest() methods is then passed all the way back to the EventManager which checks it for errors. Depending on the event, it then either returns the response object directly or simply returns a simple message string. The behaviour in case that the sendRequest() method's response contained errors also depends on the events. Some simply throw an Exception while others actually return the error messages as an array. In any case, if no Exception was thrown the EventManager returns to the Slim callback function with some kind of response.

Slim then actually responds to DispectoService with an HTTP response that follows the PSR-7 standards. Whatever the EventManager returned is JSON encoded and written to the response's body and its status set to 200, before it is send out for DispectoService to process, as shown in listing.

```
$resp = $manager->handleEvent($params, $data, $serviceinfoApi, $api);
$response->withStatus(200)->getBody()->write(json_encode($resp));
return $response;
```

Listing 13. Sending the EventManager's response

7.2 DispectoService Action (manual) Updates

In addition to sending updates automatically when an event occurs, DispectoService users also have the option to manually send certain updates to ExternalService. The user fills out the needed data and then sends the update to the provider's /actions endpoint via a POST request. As with the event updates the SirwiseAPI and ServiceinfoAPI are fetched from the dependency container and the request authorized. These along with the parsed query parameters and data are then passed to the ActionsManager class.

The ActionManager behaves largely in the same way as the EventManager. The most notable difference is that it always throws an Exception if the response object it received from the sendRequest() contained errors, and simply writes a message to response body otherwise. A bigger distinction between handling the two exists on DispectoService's side but it is beyond the scope of this Thesis.

8 Conclusion

The main aim of this thesis was to create the provider that would integrate ExternalService into DispectoService, and this was successfully accomplished. Every aspect of the desired functionality was finished, and the development was completed in the estimated time frame.

However, while the provider was finished, it has not actually been deployed yet because of shifts in Dispecto's development focus. Also, as the ExternalService does not have a test API, testing its function with the real service hasn't been possible. As such it has not yet been verified that the provider works in real use.

Despite this, some testing was necessary before the provider could be accepted as finished. To carry this out, a simple dummy application was created with Slim. This was

then run on another server and used instead of the real ExternalService in order to replicate data and errors similar to which the ExternalService might respond with when requests are sent to it. The provider was able to process both errors and successful responses correctly. Whether the errors were faulty data or timeouts, it logged them and proceeded to process the rest of the request or threw an exception as necessary depending on the response it got. With responses that contained no errors, everything also worked as expected, although fetching orders war perhaps slightly slower than expected. This may in part have been due to the slowness of the PHP server the dummy application war run on, but regardless as it was anticipated (the possibility to not do the updates on every request) it does not present a problem.

On DispectoService's side everything worked smoothly so far as they could be tested without actually using the ExternalService API. The provider was able to create new orders, update them, and also receive update from DispectoService, though these were naturally only sent to the dummy server at best. In case there were errors on the update DispectoService was able to handle them properly with the data the provider responded with. An actual request taken from the log of the older version and containing over thirty separate updates was used as one of the test responses for the dummy server, and was successfully processed by the provider, creating and updating the orders in DispectoService as per the updates.

Finally, looking into the OAuth2 authorization highlighted the need to make certain that any other providers implement it in a secure manner, for example taking CSRF into account by sending a hashed state.

References

1 Slim Documentation. [online]. <www.slimframework.com/docs/>. Accessed 1 November 2018

2 What is PRS-7 and how to use it. [online]. Dotkernel 2017. <https://www.dotkernel.com/dotkernel3/what-is-psr-7-and-how-to-use-it/> Accessed 1 November 2018

3 Slim Dependecy Injection. [online]. <https://www.slimframework.com/docs/concepts/di.html>. Accessed 15 November 2018

4 About SQLite. [online]. <https://www.sqlite.org/about.html> Accessed 20 November 2018

5 SQLite Selfcontained. [online]. <https://www.sqlite.org/selfcontained.html> Accessed 20 November 2018

6 Laravel Database Query Builder. [online]. <https://laravel.com/docs/5.5/queries> Accessed 21 November 2017

7 The Twelve Factor App. [online]. <https://12factor.net/config> Accessed 20 April 2018

8 OAuth2. [online]. <https://oauth.net/2/> Accessed 28 November 2017

9 An Intoduction to OAuth2. [online]. DigitalOcean 2014 <https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2#grant-type-resource-owner-password-credentials>. Accessed 1 December 2017

10 OAuth Security considerations [online].<https://www.oauth.com/oauth2-servers/authorization/security-considerations/>. Accessed 23 April 2018

11 Google Docs phishing email 'cost Minnesota $90,000'. [online]. BBC News 2017. <http://www.bbc.com/news/technology-39845545>. Accessed 27 April 2018

12 Johann Reinke 2016. Understanding OAuth2. [online]. <http://www.bubble-code.net/en/2016/01/22/understanding-oauth2/#Clickjacking>. Accessed 26 April 2018

13 Zach Borboa. PHP Curl Class Repository. [online].<https://github.com /php-cURL-class/php-cURL-class> Accessed 1 December 2017